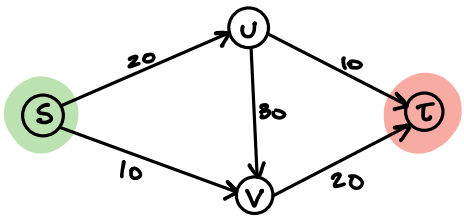
⇒ We can use graphs to model "transportation networks". That is, networks whose edges allow some sort of traffic and nodes act as switches or relies.

Examples
- Highways and interchanges
- Links and routers
- Fluid networks: pipes & junctures.

To model this, edges have **capacitres** and we assume there is a unique **source** that generates traffic and a unique **sink** nodes that absorbs it.

Example



Definition **Flow Network.**

A flow network is a directed graph $G=(V,E)$ such that:

(1) Each edge e has a non-negative capacity $C(e)$

(2) There is a single source $s \in V$

(3) There is a single sink $t \in V$

Definition **Flow**

An **s-t flow** in a **flow network** is a function f that maps each edge to a non-negative real numbers. ($f: E \to \mathbb{R}^+ \cup \{0\}$) satisfying:

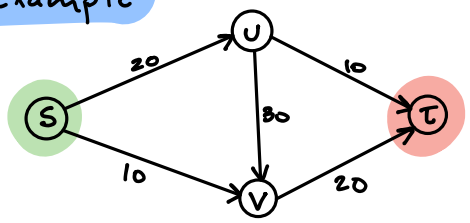
(1) $0 \leq f(e) \leq C(e) \quad \forall e \in E$ [Capacity Conditions]

(2) $\forall v \in V, \; v \notin \{s,t\}$ [Flow Conservation]

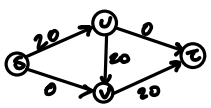
$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

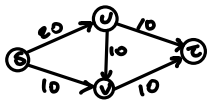
⊛ We can think of $f(e)$ as the amount of flow or traffic carried by the edge e.

Example



	f_1	f_2
(S,U)	20	20
(S,V)	0	10
(U,V)	20	10
(U,T)	0	10
(V,T)	20	20
Value	20	30





The maximum flow Problem

The **value of a flow** f is:

$$V(f) = \sum_{e \text{ out of } s} f(e)$$

Problem Given a flow network, find the flow of maximum value.

↳ This is known as the "max flow" problem

Algorithm for the Max Flow Problem

(Ford - Fulkerson 1956)

Idea

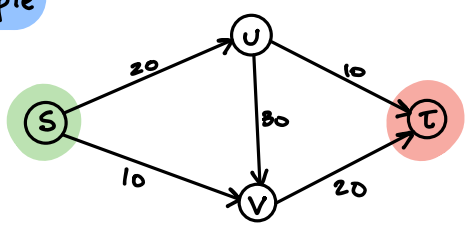
(1) Start with zero flow in every edge: $f(e)=0 \ \forall e \in E$

(2) Find path from S to T and look for edge of min capacity in this path. Add this flow value to every edge to obtain new flow.

(3) Adjust left-over capacitres

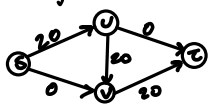
(4) Repeat (while possible)

Example

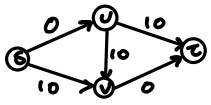


S-T Path: $S \to U \to V \to T$

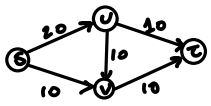
Min Capacity: 20

Flow: 

Value: 20

Left over capacities: 

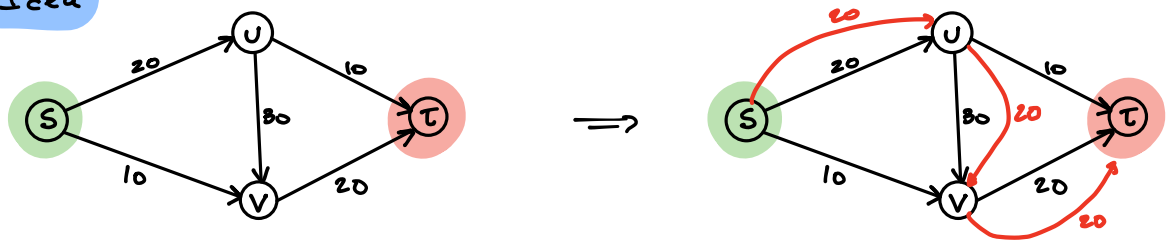
Any other s–t path? NO! Is this the maximum flow? NO!

f_2 = 

has value 30.

- We need a more systematic way of finding s–t paths

Idea

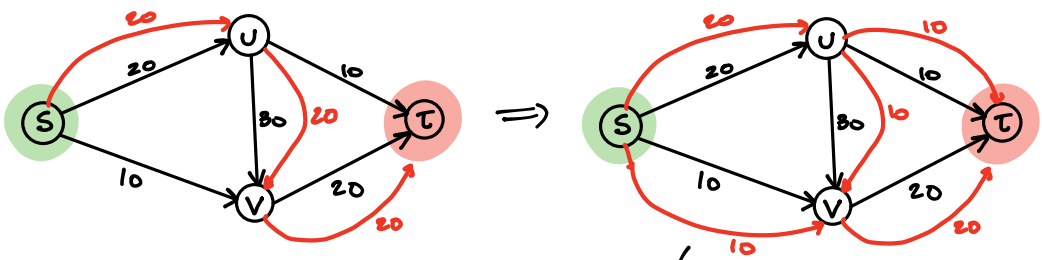


[flow is shown in red]

1) We try to find another s–t path to push more flow from s to τ.
2) No path from v to τ with additional capacity. Why?

   20 units of flow being routed from u to v to τ.
3) So, we can ask to u: can you re-route some flow through some other path?
4) u answers, Yes! I can use edge (u,τ) to re-route 10 units of flow

So,



↳This is the maximum flow.

Let us now formalize this idea.

⇒ u is asked to find a new path to τ
⇒ There is a path from s to v
⇒ There is flow "sent back" from v to u
⇒ All this combined suggests that we need to find a path but in different graph.

Definition **The residual Graph**

Given a graph $G=(V,E)$ and a flow f, we define the graph G_f as follows:

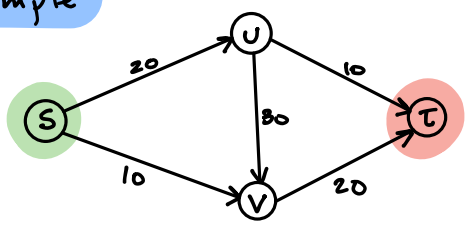
(1) G_f has the same vertex set as G

(2) $\forall e \in E$ with $f(e) < C(e)$, there are $C(e) - f(e)$ left over units of flow, so we include the edge e in G_f with capacity $C(e)-f(e)$ [forward edge]

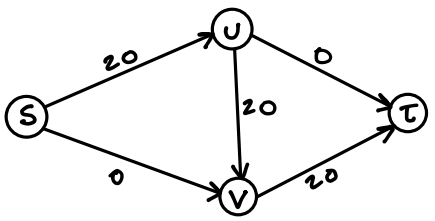
(3) $\forall e=(u,v) \in E$ with $f(e) > 0$, there are $f(e)$ units of flow going through e which could be rerouted, so we include the reverse edge (v,u) in G_R with capacity $f(e)$ [backward edge]

Example

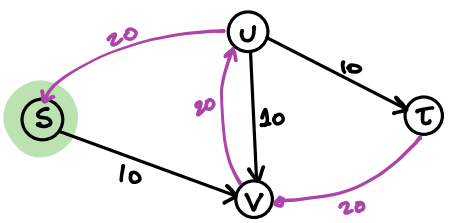
$G =$



$f =$



$G_f =$

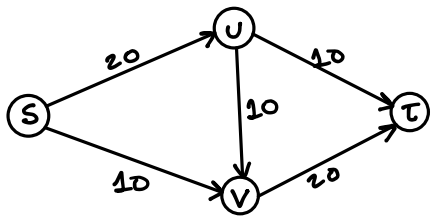


Key Insight A path in G_f can be used to improve the value of the flow.

⇒ Indeed, the path $s \to v \to u \to \tau$ tell us that we can increase the flow on the forward edges (s,v) an (u,τ) and decrease it on (u,v) (by up to 10 units in each

⇒ So with this approach, the new flow becomes:

$f_1 =$



which has a higher value.

Augmenting Paths on Residual graph

⇒ Let P be an s–t path in G_f

⇒ The **bottleneck** of the path P is given by:

$b = \text{bottleneck}(P, f) = \min$ residual capacity of any edge in G_f

⇒ We can improve the value of the flow by b, by increasing the flow in every forward edge by b and by decreasing the flow on backward edges by b.

Augment (P, f)

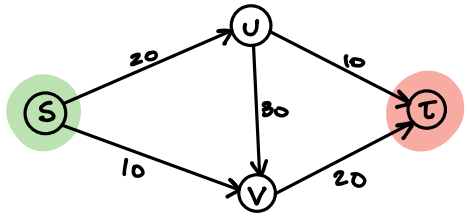
```
b = bottleneck (P, f)
for each edge (x,y) ∈ P
    if (x,y) is a forward edge
        f(x,y) = f(x,y) + b
    if (x,y) is a backward edge
        f(y,x) = f(y,x) − b.
return f
```

The Ford-Fulkerson Algorithm

Idea Start with 0 flow. Build G_f. Find s–t path and Augment to obtain new flow f_1. Buid G_{f_1}. .... until we can't find more paths to augment.

Example

G =

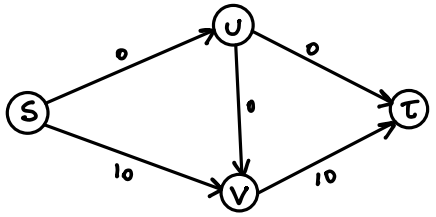


$f_0 = 0$ $G_{f_0} = G$

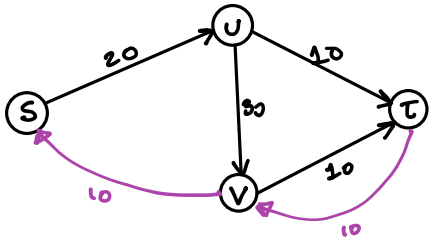
s–τ path P_0 = s → v → τ

bottleneck $(P_0) = 10$

f_1 =

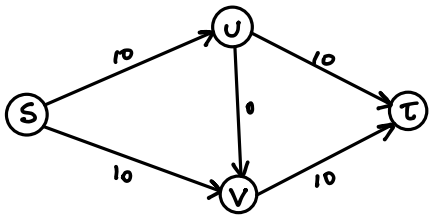


G_{f_1} =

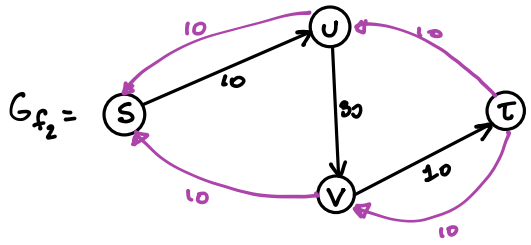


s-t path $P_1 = s \to u \to t$ bottleneck $(P_1) = 10$

$f_2 =$

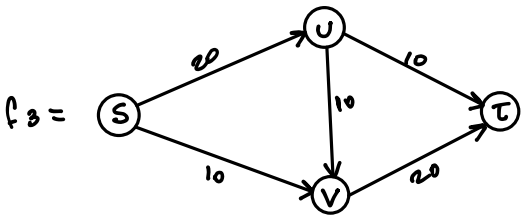


$G_{f_2} =$



s-t path $P_2 = s \to u \to v \to t$ bottleneck $(P_2) = 10$

$f_3 =$



Max-Flow-FF

Input Flow network $G = (V, E)$

Output Maximum flow f

Initialize $f(e) = 0 \quad \forall e \in E$

$G_f = G$

while there is an s-t path P in G_f.

- $f' =$ Augment (P, f)
- $f = f'$
- Build new residual graph G_f.

Output f.

Running Time ?

Correctness ?

Running time analysis

⇒ We assume for *simplicity* that all capacities are integers.

⇒ If the capacities are rational numbers, Then we can reduce to the integer case by multiplying all capacities by a suitable integer.

⇒ If the capacities are allowed to be irrational numbers Then there are ways of choosing the augmenting paths that result in FF algorithm running forever, but:

(1) This does not come often in practice

(2) Has been addressed theoretically:

Edmonds-Karp algorithm (1972)

↳ (Pick augmenting path with fewest edges)

Fact 1 In every step of the algorithm the flow and residual capacities are integers.

Fact 2 If f is a flow in G, P is augmenting in G_f and $f' = \text{Augment}(P, f)$, then:

$$v(f') = v(f) + \text{bottleneck}(P),$$

and so

$$v(f') \geq v(f) + 1.$$

Proof.

⇒ The first edge of P goes out of S.
⇒ P is simple path; i.e, no revisits.
⇒ Hence the edge out of S is a forward edge, and the flow on this edge is increased by bottleneck(P).
⇒ This implies that the value of the flow increases by exactly bottleneck(P).

Lemma. The FF algorithm performs at most $C = \sum_{e \text{ out of } S} c_e$ iterations.

Running time per iteration: $O(|V| + |E|)$

Overall running time: $O(C(|V| + |E|))$

↳ This is not polynomial on the input size!!!

Can we speed up FF algorithm?

⇒ With a simple modification: $O(|E| \cdot \log_2 C \cdot (|V|+|E|))$ already poly-time on input size!!!

⇒ Edmonds-Karp (1972): $O(|V| \cdot |E|^2)$

⇒ Dinitz (1970): $O(|V|^2 \cdot E)$

⇒ Orlin (2013): $O(N|V||E|)$ (Best strongly poly-time: independent of capacities)

⇒ Chen et al. (2022): $O(|E|^{1+o(1)} \log_2 C)$
↳ (FOCS)

Further reading:

https://en.wikipedia.org/wiki/Maximum_flow_problem#Algorithms (List of landmark improvements)

https://www.quantamagazine.org/researchers-achieve-absurdly-fast-algorithm-for-network-flow-20220608/

↳ (Cool article about Chen et al. result).

Proof of correctness

Lemma 1 If P is an augmenting s-t path in G_f, then $f' = \text{Augment}(P, f)$ is also a flow.

Definition An s-t cut of $G=(V,E)$ is a partition $(A, V \setminus A)$ where $A \subseteq V$, $s \in A$ and $t \in V \setminus A$. The capacity of the cut $(A, V \setminus A)$ is the total capacity out of A:

$$c(A, V \setminus A) = \sum_{e \text{ out of } A} c(e).$$

Lemma 2 The value of any flow f is bounded from above by the capacity of any s-t cut $(A, V \setminus A)$

$$v(f) \leq c(A, V \setminus A)$$

Lemma 3 If there is no s-t path in G_f (for some flow f), then $\exists$ an s-t cut $(A^*, V \setminus A^*)$ such that

$$v(f) = c(A^*, V \setminus A^*)$$

Lemmas 1, 2, and 3 imply the correctness of the FF algorithm.

Proof of Lemma 1.

Idea: Check capacity conditions and flow conservation constraints for f'.

$f' = \text{Augment}(P, f)$ $\quad$ $b = \text{bottleneck}(P, f)$

Capacity:

Case 1: $e \notin P$, the capacity of e does not change. Since f is a flow: $c(e) \geq f(e)$

Case 2: $e \in P$ is a forward edge.

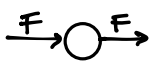
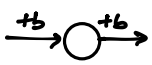
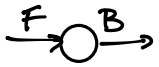
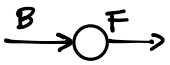
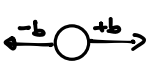
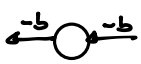
$$0 \leq f'(e) = f(e) + b \leq f(e) + (c(e) - f(e)) \leq c(e)$$

Case 3: $e \in P$ is a backward edge.

$$c(e) \geq f(e) \geq f'(e) = f(e) - b \geq f(e) - f(e) \geq 0$$

Flow conservation

Four possible cases for each vertex in P:

	in P (i.e., G_f)	in G
(1)		
(2)		
(3)		
(4)		

F: forward
B: backward

■

Proof of Lemma 2

Need to show that $\forall$ flow f and s-t cut $(A, V \setminus A)$: $\quad v(f) \leq C(A, V \setminus A)$

Idea: If k units of flow depart from $s \in A$, they need to cross $(A, V \setminus A)$ to get to $t \in V \setminus A$.

More formally.

$$f^{out}(v) = \sum_{\text{edges out of } v} f(e)$$

$$f^{in}(v) = \sum_{\text{edges into } v} f(e)$$

$$v(f) = \sum_{v \in A} f^{out}(v) - f^{in}(v)$$

because difference is 0 $\forall v \neq s$ and $f^{out}(s) = v(f)$, $f^{in}(s) = 0$

$$= \sum_{(v,w)\in A:\, v\in A} f((v,w)) - \sum_{(v,w):\, w\in A} f((v,w))$$

$$= \sum_{(v,w):\, v\in A, w\notin A} f((v,w)) - \sum_{(v,w):\, w\in A, v\notin A} f((v,w))$$

$$\leq c(A, V\setminus A)$$

Proof of Lemma 3

no s-t path in $G_f \Rightarrow \exists$ s-t cut $(A^*, V\setminus A^*)$ s.t. $v(f) = c(A^*, V\setminus A^*)$

Idea: Construct such a cut.

A^* : set of all nodes reachable from s.

$$c(A^*, V\setminus A^*) = \sum_{e \text{ out of } A^*} c(e)$$

Notice that $\forall e=(v,v')$ out of A^*, $c(e) = f(e)$, since otherwise we could reach v'. So,

$$c(A^*, V\setminus A^*) = \sum_{e \text{ out of } A^*}' f(e)$$

Likewise, $\forall e=(w,w')$ into A^*, $f(e)=0$, since otherwise we could reach w. So,

$$c(A^*, V\setminus A^*) = \sum_{e \text{ out of } A^*}' f(e) - \sum_{e \text{ into } A^*} f(e)$$

and from the proof of Lemma 2 we already know that LHS is equal to $v(f)$.